Query Processing in Containers Hosting Virtual Peer-to-Peer Nodes

Wolfgang Hoschek
CERN IT Division
European Organization for Nuclear Research
1211 Geneva 23, Switzerland
wolfgang.hoschek@cern.ch

ABSTRACT
In a Peer-to-Peer (P2P) network, a *link topology* such as a ring, tree or graph describes the link structure among autonomous nodes. A *node deployment model* defines where and how one or more partitions of the link topology are physically running, stored and accessed. Link topology and node deployment are distinct and orthogonal concepts, and hence a node deployment model need not correspond to a link topology at all. The simplest (and most common) deployment model has distinct nodes running on distinct hosts.

In this paper, we propose that nodes can also be concentrated in *node containers*, which are transparent software hosting environments that embed one or more virtual nodes. Node deployment models range from centralized to fully distributed. Virtual hosting has the potential for increased query performance (as opposed to increased scalability). Thus, we introduce the separate P2P query scope parameters *logical radius* and *physical radius*, as well as three novel query execution strategies that transparently exploit the properties of virtual hosting. The key idea is to reduce or eliminate the need for messaging between container-internal nodes and to run as few as possible queries against the database of shared nodes. Under *normal query execution* and under *collecting traversal*, a query to a container node can be efficiently answered without violating the semantics of query and scope. Under the *quick scope violating query* strategy, a query can be answered even more efficiently, by relaxing the conditions imposed by the query scope.

KEY WORDS
Peer-to-Peer Network, Virtual Hosting, Query Processing

1. Introduction

In a large distributed system such as a Peer-to-Peer (P2P) file sharing system [1, 2] or a Grid [3], it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. Other examples are a (worldwide) service discovery infrastructure for a multi-national organization, the Domain Name System (DNS), the email infrastructure, a monitoring infrastructure for a large-scale cluster of clusters, or an instant messaging and news service. For example, the European DataGrid (EDG) [4, 5] is a global software infrastructure that ties together a massive set of globally distributed organizations and computing resources for data-intensive physics analysis applications, including thousands of network services, tens of thousands of CPUs, WAN Gigabit networking as well as Petabytes of disk and tape storage [6].

An enabling step towards increased Internet and Grid software execution flexibility is the *web services* vision [7, 8, 9] of distributed computing where programs are no longer configured with static information. Rather, the promise is that programs are made more flexible and powerful by querying Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks. Services can advertise themselves and related metadata via such databases, enabling the assembly of distributed higher-level components.

In support of this vision we have introduced the *Web Service Discovery Architecture (WSDA)* [10] and given motivation and justification [11] for the assertion that realistic ubiquitous service and resource discovery requires a rich general-purpose query language such as XQuery [12] or SQL [13]. Based on WSDA, we introduced the *hyper registry* [14], which is a centralized database (node) for discovery of dynamic distributed content.

However, in an Internet discovery database system, the set of information tuples in the universe is partitioned over one or more distributed nodes (peers), for reasons including autonomy, scalability, availability, performance and security. It is not obvious how to enable powerful discovery query support and collective collaborative functionality that operate on the distributed system as a whole, rather than on a given part of it. Further, it is not obvious how to allow for search results that are fresh, allowing time-sensitive dynamic content.

It appears that a Peer-to-Peer (P2P) database network may be well suited to support dynamic distributed database search, for example for service discovery. The overall P2P idea is as follows. Rather than have a centralized database, a distributed framework is used where there exist one or more autonomous database nodes, each maintaining its own data. Queries are no longer posed to a central database; instead, they are recursively propagated over the network to some or all database nodes, and results are collected and send back to the client.

Consequently, we devised the WSDA based *Unified Peer-to-Peer Database Framework (UPDF)* [15] and

its associated *Peer Database Protocol (PDP)* [16], which are unified in the sense that they allow to express specific applications for a wide range of data types (typed or untyped XML, any MIME type [17]), node topologies (e.g. ring, tree, graph), query languages (e.g. XQuery, SQL), query response modes (e.g. Routed, Direct and Referral Response) and pipelining characteristics. In the UPDF framework, an *originator* sends a query to an *agent* node, which evaluates it, and forwards it to select *neighbor nodes*.

In a Peer-to-Peer (P2P) network, a *link topology* such as a ring, tree or graph describes the link structure among autonomous nodes. For example, in a worldwide service discovery system, a link topology can tie together a distributed set of administrative domains, each hosting a registry node holding descriptions of services local to the domain. Figure 1 depicts some example topologies, covering the spectrum from centralized models to fine-grained fully distributed models.

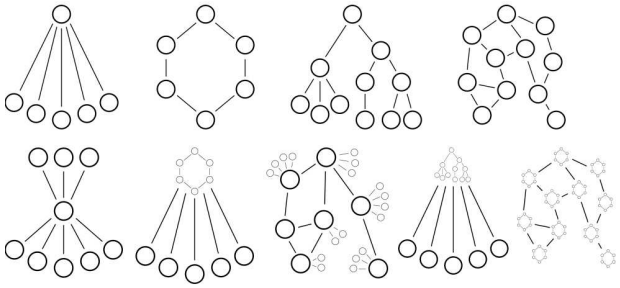

Figure 1. Example Link Topologies [18].

A *node deployment model* defines where and how one or more partitions of the link topology are physically running, stored and accessed. Link topology and node deployment are distinct and orthogonal concepts, and hence a node deployment model need not correspond to a link topology at all. The simplest (and most common) deployment model has distinct nodes running on distinct hosts.

In this paper, we propose that nodes can also be concentrated in *node containers*, which are transparent software hosting environments that embed one or more virtual nodes. Node deployment models range from centralized to fully distributed. Virtual hosting has the potential to increased the query performance (as opposed to increased scalability). Thus, we introduce the separate P2P query scope parameters *logical radius* and *physical radius*, as well as three novel query execution strategies that transparently exploit the properties of virtual hosting. The key idea is to reduce or remove the need for messaging between container-internal nodes and to run as few as possible queries against the database of shared nodes. Under *normal query execution* and under *collecting traversal*, a query to a container node can be efficiently answered without violating the semantics of query and scope. Under the *quick scope violating query* strategy, a query can be answered even more efficiently, by relaxing the conditions imposed by the query scope.

The remainder of this paper is organized as follows. Section 2. motivates and describes the use of containers for virtual node hosting. Section 3. discusses in detail the three query execution strategies. Finally, Section 4. summarizes and concludes this paper.

2. Containers for Virtual Node Hosting

A node link topology can be deliberately arranged and exploited by applications. For example, in an attempt to explicitly exploit topology characteristics, a virtual organization of a Grid may deliberately organize global, intermediate and local job schedulers into a tree-like topology. Correct and efficient operation of scheduling may involve queries with a neighbor selection policy that selects all child nodes and ignores all parent nodes. For the scheduling query, it is irrelevant where the nodes are running, and where and how nodes (tuples, service descriptions) are stored. What matters is that the query traverses a tree.

A link topology is purely a logical construct. It does *not* describe where and how this link information is stored and accessed. This is defined by a *node deployment model*, which defines where and how one or more partitions of the graph are running, stored and accessed.

We argue that link topology and node deployment are distinct and orthogonal concepts, and hence a node deployment model need not correspond to a link topology at all. Consider the analogy to the WWW: The WWW is a graph of HTML pages. Vertices are established through embedded HTML hyperlinks. The graph topology is, by definition, insensitive to how and where HTML pages are physically stored and served (on which hosts, URL paths, and web server technologies). The topology remains identical, no matter whether all pages of the universe are served by a single large dynamic web server or any kind of worldwide federation of static web servers.

The simplest (and most common) deployment model has distinct nodes running on distinct hosts. However, we propose that nodes can also be concentrated in central places called node containers. A *node container* is a transparent software hosting environment that embeds one or more nodes, as depicted in Figure 2. The set of all nodes in the universe is partitioned over one or more node containers. A container can be a special-purpose program that *behaves as if* it were a network of nodes (*virtual hosting*). A well-known example for virtual hosting is web serving. A web server can serve millions of static or dynamic pages from an essentially infinitely large name space of URLs (nodes). To the outside world, the server is invisible, and each URL (node) can be seen as a separate service (having a name or address, HTTP network protocol, TLS security, etc). Of course, internally just one or a few processes are used to implement such virtual hosting. A general-purpose container, on the other hand, can run each node (or each request to a node) in an independent process or thread (*physical hosting*). For example, virtual and physical hosting is used in Java servlet [19] and Enterprise Java Beans technology [20].

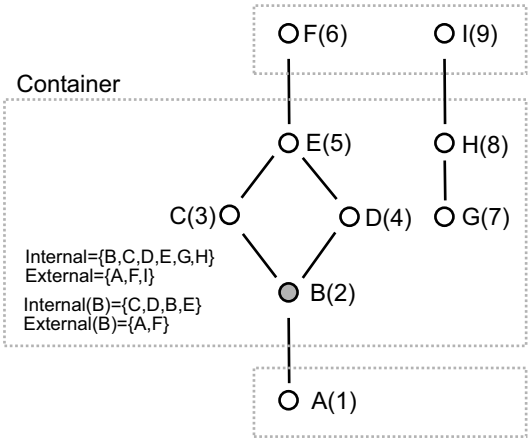

Figure 2. Node Containers.

In any case, the container is invisible to the outside world. Hosted nodes still appear and behave like any other node. In our case, this means that a hosted node has a service link and description, and it supports publication, queries, etc. via the operations and network protocols advertised by the service description. The fundamental difference to classic database architectures is that in the latter there exists no deployment transparency. As an extreme example of virtual hosting, one could imagine a hypothetical relational database system that exposes each individual tuple as a network service, supporting direct network connections to the tuple to answer queries against its column values. Conceptually, we can say that every node runs within a container, even if the container holds only a single node. A remote client may ask for the dynamic creation of a virtual or physical node by means of a node factory interface. Now we are in the position to define a *node deployment model* as being a description of the set of containers physically implementing a given link topology.

Several node deployment models can be envisaged, ranging from coarse to fine grained, as well as arbitrary mixtures. For example, in a centralized deployment scenario, the entire global graph of, say 10^8, nodes may be accessible through a single container, with all nodes (service interfaces) being handled by a single process on a single host. In a slightly less central scenario, the same graph may be partitioned among ten organizations, each with a central container as described above. On the fully distributed end of the spectrum, each node may run on a distinct box, storing its own tuples (including neighbor descriptions) in a local registry. In all but the first case, there neither exists a single grand monolithic database nor a single owner and provider of information.

There are two primary motivations why concentrating nodes may be useful. First, for reasons including central control, reliability, continuous availability, maintainability, security, accounting and firewall restrictions on incoming connections for hosts. These reasons are important, but we do not delve into them any further. Similarly, we do not consider physical hosting any further. Instead, we focus on the second motivation, which is the potential of virtual hosting for increased performance (as opposed to increased scalability).

In any kind of P2P network, a node has a database or some kind of data source against which queries are applied. In a P2P network for service discovery, this database happens to be the publication database. Discussion in this section is applicable to any kind of P2P network, while the examples illustrate service discovery.

If many container nodes reside on the same host, in the same process and store their tuples (e.g. node service descriptions, Gnutella file indexes) in the same database, query support is potentially much more efficient. The query engine can run on "big iron". The database may fit in its entirety in a main memory buffer. Network communication between remote nodes can be replaced with local loop-back connections, inter-process communication or even direct function calls. To compute the full query result set for all container nodes, it may perhaps be sufficient to execute just one or a few batch queries against the shared database, instead of many small queries against separate databases. Intuitively it seems that the smaller nodes are, the more performance can be gained through virtual hosting. For example, consider a network with millions of small registry nodes spread all over the world, each holding just some ten tuples. Perhaps searching would be much more efficient if the nodes and their databases were just partitioned across a few, say a hundred, powerful node containers.

Consider the three example containers depicted in Figure 2. The central container has six internal nodes `(B,C,D,E,G,H)` and three external nodes `(A,F,I)`. External nodes belong to other containers. *Internal links* connect nodes within the container. *External links* connect internal with external nodes. A hop is said to be *logical* if it travels along an internal or external link. A hop is said to be *physical* if it travels along an external link. Intuitively it is clear that traversing an internal link is much cheaper than traversing an external link. Accordingly we propose to distinguish the separate scope parameters *logical radius* and *physical radius*. For example, a user can specify that a query should reach very far, say a logical radius of 100 hops. To ensure that this query does not burden all nodes in the universe, the user can specify that it should touch at most three containers on any given path (physical radius).

3. Container Query Processing

In this section, we propose three query execution strategies. A query to a node of a container can be efficiently answered without violating the semantics of query and scope (*normal query execution*, *collecting traversal*). Even more efficiently, it can be answered by relaxing the conditions imposed by the query scope (*quick scope violating query*). Let us look at these three strategies in more detail.

Normal Query Execution. Clearly a container can answer a query like any normal node via the execution plans proposed in our prior studies [15]. Recall the template execution plan, as depicted in Figure 3, and the

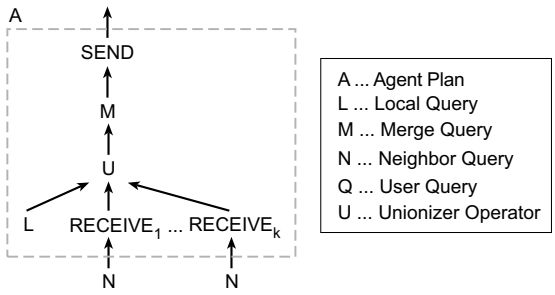

Figure 3. Template Execution Plan [15].

specific plans for queries that are either recursively partitionable or not. As an optimization, network communication between internal nodes can be replaced with local loop-back connections, inter-process communication or direct function calls. For example, our Peer Database Protocol [16] is built upon the BEEP application-level network protocol framework [21, 22]. Since BEEP can be mapped to several underlying reliable transport layers (TCP is merely the default), a container can plug in an in-process transport mapping, yet continue to use the same messaging code base.

Collecting Traversal. To answer queries, a container can use a strategy we propose to call *collecting traversal*. Here the goal is to remove the need for any internal messaging and to run as few as possible queries against the database of shared nodes. To ensure that query semantics are fully preserved, the fact is exploited that queries in our query model are defined over a single virtual set of tuples (service descriptions). The query model allows generating this set of tuples in any arbitrary way.

The strategy works as follows. When a container node receives an external query, it takes over the work for the other internal nodes. In the first phase, it collects preparatory data. In the second phase, the query is executed. The first phase collects the internal and external nodes that are reachable from the start node. In other words, one traverses the container from the given start node, following the path that the query *would* touch. Along the way, the (keys of) internal nodes and external nodes are collected.

Consider the example from Figure 2. The keys of the six internal nodes are `(2,3,4, 5,7,8)` whereas the keys of the three external nodes are `(1,6,9)`. The originator sends a query to the start node `B`. The node has a database of `tuples(B)`=$\{$`1,3,4`$\}$. The internal nodes reachable from `B` are `internal(B)`=$\{$`3,4,2,5`$\}$, and the reachable external nodes are `external(B)`=$\{$`1,6`$\}$. The tuples contained in the internally reachable nodes are `tuples(internal(B)) = UNION (tuples(3), tuples(4), tuples(2), tuples(5))` = $\{$`1, 2,3,4,5,6`$\}$.

According to query type, the node chooses an execution plan, and executes it. However, the local query L is executed against the tuples of the internally reachable nodes `tuples(internal(B))` rather than against the tuples of the database `tuples(B)`. Similarly, the plan forwards the query to the nodes `external(B)` rather than to the neighbors obtained from `B`'s neighbor selection. Scope semantics are preserved by explicitly applying the relevant rules of scope parameters during node traversal (e.g. radius pruning and neighbor selection).

The net effect is that the local query `L` and the merge query `M` are batched. That is, they are applied once over a large set, instead of many times over a small set. The nodes of a container are stored in a single database (table), for example as depicted in Table 1. The table is not normalized for clarity of exposition. Collecting nodes is particularly fast if the neighbor selection policy is simple and the database fits into main-memory. For example, it is certainly possible to have a data structure that allows quickly traversing the database table. Further, internal messaging overhead is eliminated altogether. To summarize, if the neighbor selection policy is applied at each node, and scope parameters such as radius are observed, one can emulate normal query execution, but in a way that is more efficient.

Node ID	Service	Is external?	Tuples
1	A	True	null
2	B	False	3, 4
3	C	False	2, 5
4	D	False	2, 5
5	E	False	3, 4, 6
6	F	True	null

Table 1. Node Table of Container.

Note that the radius is not defined to be *the number* of hops a query is allowed to travel on any given path. Rather, it is more weakly defined to be the maximum number of hops a query is allowed to travel on any given path. In other words, it is not guaranteed that a query takes the shortest path from the agent to any given node, thereby covering a total maximum of nodes. There are two reasons for this kind of definition. First, a node may choose to decrease the radius by any value it sees fit in order to reduce resource consumption or to prevent system exploitation. Second, loop detection and unpredictable timing in distributed systems can lead to a phenomenon we propose to call *greedy radius pruning*. Recall that the very same query may arrive at a node multiple times, along distinct routes, perhaps in a complex pattern. Traveling `N` hops decreases the radius of a query by `N`. If the query first arrives via a route with many (fast) hops, and later arrives again via a route with few (slow hops), the second arrival will be detected as a loop and rejected. However, the successfully forwarded (first) query continues to travel less hops than theoretically possible considering the (larger) radius of the second query. If the second query had arrived first, the query would have been able to travel further and potentially collect more matching results. Propagating a query to all neighbors concurrently may somewhat increase query coverage, in particular in homogenous LANs.

Traversal of internal nodes via depth-first search is

inappropriate because it leads to greedy radius pruning with high probability, in particular if a container holds a large number of nodes with a complex internal topology. In practice, this means that even though a user may have specified a theoretically large enough logical radius, it is unlikely that an incoming query will ever forward beyond the current container. It is in the nature of depth-first search that it is unlikely that an external link is reached along a short internal route. Rather, it is likely that it is reached along one of the longest possible internal routes. Within a container, greedy pruning can be eliminated by traversal using breadth-first search. This ensures that the shortest path to external links is always found, despite loop detection pruning. Put another way, loop detection is conditioned to prune only paths longer than the shortest path. The net effect is that external nodes receive a meaningful logical radius scope parameter on query forward. The pseudo-code in Figure 4 computes the internal and external nodes of a given entry node, using breadth-first search.

```
FUNCTION collectingTraversal(startNode, logRadius) {

internal = {}, external = {}
done = {}, todo = {startNode}
while (size=size(todo)) > 0 and logRadius >= 0
  logRadius = logRadius - 1
  for i := 1 to size
    node = first element of todo
    remove first element from todo
    done = done UNION {node}
    internal = internal UNION tuples(node)
    if logRadius >= 0 then
      for each neighbor n IN select(neighbors(node))
        if n is internal && not contained in todo &&
          not contained in done
            then Append n to todo
        endif
        if n is external &&
           not (n,any radius) contained in external
           then external=external UNION
                                    {(n, logRadius)}
        endif
      endfor
    endif
  endfor
endwhile
Return (internal, external)
}
```

Figure 4. Collecting Traversal.

Quick Scope Violating Query. *Normal query execution* and *collecting traversal* preserve query and scope semantics. If no query scope is given, or if it is acceptable to ignore or alter scope semantics, query execution can be optimized further using the strong technologies of centralized (relational) database architectures. For example, internal graph traversal can be eliminated altogether. The strategy works as follows. According to query type, the node chooses an execution plan, and executes it. However, the local query `L` is executed against the union of *all* tuples of the container `(1-9)` rather than against `B`'s database `tuples(B)`. Similarly, the plan forwards the query to the external nodes selected from the union of *all* external nodes of the container `(1,6,9)` rather than to the immediate neighbors of `B`.

The net effect is that the local query `L` and the merge query `M` are batched. That is, they are applied once over a large set, instead of many times over a small set. The same holds for neighbor selection. Determining all tuples of the container requires no time at all because they are, of course, stored in the same database (table). Computing all external nodes is cheap as well. Scope-violating queries are answered using the strong technologies of centralized (relational) database architectures. Consequently, they are highly efficient, at the expense of ignoring or altering scope semantics.

For example, nodes `(7,8,9)` should never be considered, as they are not directly or indirectly connected to B. Further, it is unclear what logical radius should be assigned on query forward to external nodes. Computing the correct logical radius would essentially degrade performance down to the performance of *collecting traversal*. It appears that the least bad choice is to decrease the logical radius by one on external forward. Note that query semantics are still preserved. The query is just fed a larger than necessary set of tuples (service descriptions). In practice, this may be tolerable for a significant fraction of use cases.

4. Conclusions

Link topology and node deployment are distinct and orthogonal concepts, and hence a node deployment model need not correspond to a link topology at all. The simplest (and most common) deployment model has distinct nodes running on distinct hosts. A *node container* is a transparent software-hosting environment that embeds one or more nodes. The set of all nodes in the universe is partitioned over one or more node containers. A container can be a special-purpose program that *behaves as if* it were a network of nodes (*virtual hosting*). A well-known example for virtual hosting is web serving. Hosted nodes still appear and behave like any other node. In our case, this means that a hosted node has a service link and description, and it supports publication, queries, etc. via the operations and network protocols advertised by the service description. Node deployment models range from centralized to fully distributed. Virtual hosting has the potential for increased performance (as opposed to increased scalability). For example, consider a network with millions of small registry nodes spread all over the world, each holding just some ten tuples. Perhaps searching would be much more efficient if the nodes and their databases were just partitioned across a few, say a hundred, powerful node containers.

Internal links connect nodes within a container. *External links* connect internal with external nodes. The separate scope parameters *logical radius* and *physical radius* are distinguished. A query to a container node can be efficiently answered without violating the semantics of query and scope (*normal query execution*, *collecting traversal*). Even more efficiently, it can be answered by relaxing the conditions imposed by the query scope

(*quick scope violating query*).

The goal of *collecting traversal* is to remove the need for any internal messaging and to run as few as possible queries against the database of shared nodes. To ensure that query semantics are fully preserved, the fact is exploited that queries in our query model are defined over a single virtual set of tuples. The query model allows generating this set of tuples in any arbitrary way. The first phase collects the internal and external nodes that are reachable from the start node. The local query is executed against the tuples of the internally reachable nodes. The query is forwarded to the reachable external nodes. Scope semantics are preserved by explicitly applying the relevant rules of scope parameters during node traversal. Traversal of internal nodes via depth-first search is inappropriate because it leads to greedy radius pruning with high probability. Breadth-first search should be used instead.

If no query scope is given, or if it is acceptable to ignore or alter scope semantics, a query can be answered with the *quick scope violating query* strategy, using the strong technologies of centralized (relational) database architectures. Internal graph traversal is eliminated altogether. The local query, the merge query and neighbor selection are applied once over a large set, instead of many times over a small set.

References

[1] Gnutella Community. Gnutella Protocol Specification v0.4. dss.clip2.com/GnutellaProtocol04.pdf.

[2] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[3] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int'l. Journal of Supercomputer Applications*, 15(3), 2001.

[4] Ben Segal. Grid Computing: The European Data Grid Project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, France, October 2000.

[5] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *1st IEEE/ACM Int'l. Workshop on Grid Computing (Grid'2000)*, Bangalore, India, December 2000.

[6] Large Hadron Collider Committee. Report of the LHC Computing Review. Technical report, CERN/LHCC/2001-004, April 2001. http://cern.ch/lhc-computing-review-public/Public/Report_final.PDF.

[7] Wolfgang Hoschek. *A Unified Peer-to-Peer Database Framework for XQueries over Dynamic Distributed Content and its Application for Scalable Service Discovery*. PhD Thesis, Technical University of Vienna, March 2002.

[8] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steve Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, January 2002. http://www.globus.org.

[9] P. Cauldwell, R. Chawla, Vivek Chopra, Gary Damschen, Chris Dix, Tony Hong, Francis Norton, Uche Ogbuji, Glenn Olander, Mark A. Richman, Kristy Saunders, and Zoran Zaev. *Professional XML Web Services*. Wrox Press, 2001.

[10] Wolfgang Hoschek. The Web Service Discovery Architecture. In *Proc. of the Int'l. IEEE/ACM Supercomputing Conference (SC 2002)*, Baltimore, USA, November 2002. IEEE Computer Society Press.

[11] Wolfgang Hoschek. A Data Model and Query Language for Service Discovery. Technical report, DataGrid-02-TED-0409, April 2002.

[12] World Wide Web Consortium. XQuery 1.0: An XML Query Language. *W3C Working Draft*, December 2001.

[13] International Organization for Standardization (ISO). Information Technology-Database Language SQL. *Standard No. ISO/IEC 9075:1999*, 1999.

[14] Wolfgang Hoschek. A Database for Dynamic Distributed Content and its Application for Service and Resource Discovery. In *Int'l. IEEE Symposium on Parallel and Distributed Computing (ISPDC 2002)*, Iasi, Romania, July 2002.

[15] Wolfgang Hoschek. A Unified Peer-to-Peer Database Framework for Scalable Service and Resource Discovery. In *Proc. of the 3rd Int'l. IEEE/ACM Workshop on Grid Computing (Grid'2002)*, Baltimore, USA, November 2002. Springer Verlag.

[16] Wolfgang Hoschek. A Unified Peer-to-Peer Database Protocol. Technical report, DataGrid-02-TED-0407, April 2002.

[17] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. *IETF RFC 2045*, November 1996.

[18] Nelson Minar. Peer-to-Peer is Not Always Decentralized. In *The O'Reilly Peer-to-Peer and Web Services Conference*, Washington, D.C., November 2001.

[19] Java Community Process. Java Servlet 2.3 Specification. jcp.org/aboutJava/communityprocess/final/jsr053.

[20] Java Community Process. Enterprise Java Beans Specification. java.sun.com/products/ejb/docs.html.

[21] Marshall Rose. The Blocks Extensible Exchange Protocol Core. *IETF RFC 3080*, March 2001.

[22] Marshall Rose. Mapping the BEEP Core onto TCP. *IETF RFC 3081*, March 2001.